

Method of Accessing Data and Logic on Existing Systems through Dynamic Construction of Software Components

Cross Reference to Related Applications

This application is a continuation-in-part application of currently pending applications serial number 09/385,903 filed August 30, 1999.

Background of the Invention

The present invention relates generally to the field of computer software development tools. It relates in particular to the generation of executable software components based on existing structured data in non-component-based languages to access the contents of the structured data at runtime.

Much of the new software development projects are moving to object-oriented development environments from the existing procedural, structured development environments. More progressive development environments also exploit the use of component-based design, which is an extension of object-oriented programming techniques. Developers have confirmed what computer science predicted a couple of decades ago – namely that object-oriented and component-based design yields more maintainable and reusable code. Given the complexity of modern network-based or “e-Business” applications, there is a strong desire among businesses to utilize modern development languages and techniques to manage the complexity.

One of the most popular programming languages for the development of network-centric applications is Java®. Java, developed by Sun Microsystems, is an

1 object-oriented, multithreaded, platform independent programming language. The
2 design of the language, coupled with the rich set of available class libraries along with
3 the specification of a component model, called JavaBeans®, within the language make
4 it well-suited to develop applications that interact with other business applications or
5 customers via the World Wide Web.

6 One component framework built by Sun Microsystems for the Java environment
7 is the JavaServer Pages ("JSP") framework. A JSP document contains a mixture of a
8 text-based markup language (usually HTML but it can also be an XML-derived
9 grammar) with small segments of Java code and JSP-specific tags. This all-text
10 document is dynamically translated into Java source code and compiled dynamically
11 into a Java binary file that is suitable for execution within a Java Virtual Machine
12 environment. At present, JSPs are used primarily as a technique to separate the
13 business logic in server-based Java applications ("servlets") from the presentation of
14 the data to a web browser in HTML.

15 As companies create a presence on the Internet for business-to-business
16 transactions and business-to-customer transactions ("web transactions"), there is a
17 strong desire to use the design methodologies and language support for object-
18 orientation and component based design. However, most of the application logic and
19 data that are required to complete a web transaction are contained within existing
20 procedural code. This procedural code was developed using languages such as
21 COBOL and PL/I that are poorly-suited to network-based programming inherent in
22 building web transactions. Furthermore, this business-critical procedural code was
23 developed and is maintained by programmers who, by and large, do not have the skills

1 in object-oriented design and development and component-based software
2 architectures. Even for companies that have strong programmer skills in modern
3 programming languages and software development techniques, there are substantial
4 barriers in facilitating web transactions from an existing procedural code base. It is
5 inherently cost-prohibitive to recreate the business applications in an object-oriented
6 programming language such as Java. Even if the funding could be justified, the time-
7 to-market is too long to meet business goals. Most companies have recently completed
8 reinvesting in their existing procedural code base to ensure that it is compliant for date
9 processing in the year 2000 ("Y2K compliant"). This indicates a strong willingness by
10 corporations to continue to gain benefit from the existing base of code instead of simply
11 replacing it.

12 Companies wanting to engage in e-Business face two conflicting desires. The
13 first is the desire to manage software complexity inherent in web transactions with
14 modern object and component-based programming languages. The second is the
15 desire to continue to leverage business logic and data found in the existing applications
16 developed with procedural programming languages and techniques.

17 One popular technique to bridge existing systems and new systems is the use of
18 the Extended High-Level Language Application Programming Interface ("EHLLAPI" or
19 "screen scraping"). The term "screen scraping" is indicative of how the solution
20 operates. Applications are developed, presumably using modern development
21 languages, to mimic the behavior of a human terminal user. The software reads
22 ("scrapes") the textual information from the terminal display and interprets the data
23 much as a human would. Similarly, feedback is provided to the existing system by the

1 new application simulating the entry of keystrokes by a human. This simulation of
2 human terminal interaction with software is problematic. The nature of such
3 applications is highly process-oriented, and at first glance, would appear to be easily
4 automated in software. However, humans are vastly more capable of intelligent screen
5 navigation than is present software. Humans are also able to recognize out-of-context
6 situations, such as error screens or indications that the host computer is unavailable.
7 The design of screen navigation logic and error detection and recovery logic is often as
8 complicated as the network programming and complexities of web transactions that the
9 solution was trying to avoid. The screen scraping solution often runs into problems of
10 scalability and performance. The conventional or legacy software systems were
11 designed to interact with dozens or perhaps hundreds of corporate terminal users.
12 When these legacy software systems become the conduit for thousands or millions of
13 Internet users, the architecture often fails to scale to meet the demand.

14 Another technique to bridge between web transactions and existing systems is to
15 access the persistent data directly. Such a bridge is described in U.S. Patent No.
16 6,081,808 (IBM patent) wherein the data from the legacy system is directly accessed,
17 but the business logic of the legacy application is not applied. While workable
18 solutions exist for accessing databases from client/server or network-centric
19 applications, this technique completely bypasses the business rules that apply to the
20 raw data. Most data processing systems of any complexity are more than the
21 presentation of stored data. They use software applications to manipulate the data into
22 usable information. Therefore, a great need exists for a method of accessing and
23 incorporating the complexity of legacy processing system applications with the

1 accessing of the legacy data while dynamically manipulating that legacy logic and data
2 for use on present day object-oriented environments.

3 Other solutions to bridging the gap between modern multi-tier object-oriented
4 network-centric applications and the existing procedural legacy applications exist in
5 middleware products such as IBM's® MQSeries or the distributed features in IBM's
6 CICS product. Even with these middleware products, there is a need for programmers
7 to understand the differences in the underlying binary data formats of the legacy
8 computing platform and the architecture of the platforms on which the object-oriented
9 systems run. When data is passed between platforms, data conversion must be
10 programmed to account for the architectural differences. For example, for each field in
11 each record that is a text field, such as a person's name, if the source information
12 comes from an IBM S/390 mainframe system, each letter will have to be converted from
13 the internal mainframe EBCDIC encoding to the Java Unicode encoding for alphabetic
14 letters. There is also an issue with the ordering of bytes in different computer
15 architectures. For example, the number 6,091,960 is represented on an Intel
16 Pentium® machine architecture as the hexadecimal number B8F45C00 whereas the
17 same number on an IBM S/390 mainframe is represented internally as 005CF4B8.
18 (Note that the ordering of the hexadecimal pairs of digits are backward with respect to
19 each other). The code to translate all of the internal data type representations is error-
20 prone due to the need for programmers to understand all of the machine architectures
21 they will encounter and the low, bit-level manipulation that is required for many of the
22 translations. The present invention provides a unique solution to the problem of
23 bridging web transactions to existing applications.

Summary of the invention

An objective of this invention is to dynamically construct a software component from the contents of the binary data contained within a running procedural application on an existing system. The resulting software component is able to be programmed using modern object-oriented and component-based languages and processes to facilitate the development of systems to be used, for example, for e-Commerce.

Another objective of this invention is to transition between procedural skills and languages into object and component-based languages in order to leverage the skills of two important, but disparate, groups of programmers present within most corporate environments. Procedural programming began to emerge as the programming technique of choice in the late 1960's and early 1970's and still represents a large percentage of the programming skills in the marketplace. Procedural code and the way that it typically interacts with the data – in the form of record-oriented input and output – represents the bulk of the software currently in production. The businesses have encapsulated the algorithms and processes, called “business rules”, in the form of this procedural software. The result of this invention is to facilitate the reuse of existing procedural code that represents a corporation's business rules, and possibly their competitive advantage in the marketplace, and integrate that valuable resource into modern web transactions that are developed using object-oriented and component-based development techniques.

The present invention is comprised of three primary application frameworks. The first framework (“workbench”) takes source record layout definitions in the programming language in which they were originally developed and generates a

1 language-neutral and machine architecture-neutral representation of the structure and
2 fields within that record layout. For example a COBOL copybook, a Pascal record or a
3 PL/I record definition is copied from the existing application by the programmer using
4 commonly-available text editing software such as ISPF/PDF on a mainframe system or
5 vi on a Unix system to the system hosting the workbench framework. The programmer
6 uses the workbench software that is part of the present invention to pass that textual
7 source code to a lexical analyzer and parser that is part of the workbench framework of
8 the present invention. The workbench will generate a language-neutral representation
9 of the fields within the record layout along with the structural context of the fields. The
10 structural context means the hierarchical structure within the record, for example that
11 the fields are contained within a homogeneous array of fields or that the fields are part
12 of a collection of heterogeneous fields in a substructure of the whole record structure.
13 This part of the present invention creates the data about the data in the copybook
14 ("metadata"). The metadata is saved in a repository on some persistent storage
15 medium such as a database or within a file system. **Figure 8** demonstrates the flow of
16 information between the existing source code for an application and the workbench.
17 This process is a one-time preparatory step and is "offline" with respect to the
18 application as it runs as illustrated in **Figure 11**.

19 The second application framework is a programming interface for the legacy
20 application to communicate with the runtime environment. This is comprised of a set of
21 program subroutine calls ("Application Programming Interfaces" or "APIs") that
22 implement an open/close/read/write metaphor. This programming metaphor,
23 demonstrated in **Figure 9**, is typical of procedural programming. There is an

1 implementation of the API set that is part of the present invention for each programming
2 language. Examples include, but are not limited to, COBOL, COBOL under CICS, PL/I,
3 Pascal, or C. **Figure 10** shows the implementation of the programming interface in the
4 C programming language. This framework of the present invention is responsible for
5 delivering two pieces of information to the component runtime framework of the present
6 invention. 1) The name of or numeric reference to the record layout, which is used to
7 identify the metadata created by the workbench, and 2) a reference to the binary data
8 contained within the record layout at the time the programming call to read or write
9 data. The reference to the binary data is most likely a memory address (a "pointer") as
10 implemented in most programming languages. The implementation of the APIs
11 includes calls to an underlying interprocess communications mechanism such as FIFOs
12 in the Unix operating system or named pipes in personal computing operating systems
13 such as Windows NT or OS/2. The interprocess communications mechanism can also
14 be an inter-machine networking protocol such as TCP/IP sockets or APPC over SNA
15 networks. The specific implementation is hidden via a programming abstraction from
16 the legacy applications programmer utilizing the API framework. This second
17 application framework is co-located on the same computer platform as the customer's
18 existing procedural code. As a result, it is designed to be a minimal implementation of
19 programming logic and essentially becomes a bridge between the customer code and
20 the sophisticated processing and complexity inherent in the third application framework
21 described below.

22 The third application framework of the present invention is the "component
23 runtime environment". While the legacy application is running, it will have been

1 programmed to either read or to write a record using the APIs that are part of the
2 present invention and are described above (an example of which appears as **Figure**
3 **10**). The process of reading or writing a record from the legacy application perspective
4 is conceptually the same, but the implementation is different, so they are discussed
5 separately. When the host writes a record using the APIs supplied by the present
6 invention, the runtime environment component creates a new instance of the
7 component identified for this record. The binary contents from the legacy application
8 are loaded into the software component. This automates the process that would
9 otherwise be tedious and error-prone programming to translate the bit patterns
10 appropriate for source platform into the bit patterns appropriate for the destination
11 platform. The component is now suitable for object-oriented programming by
12 applications that utilize the present invention to gain access to dynamic data that may
13 have been created by a combination of persistent data and business logic within a
14 legacy application.

15 The process of the legacy application reading the data from the component of
16 the present invention is similar. The legacy application reads a record using the APIs
17 supplied by the present invention (for example using the `IxsRead(...)` example from
18 **Figure 10**. The API will identify the record to be read from the runtime environment.
19 The API call pauses execution of the legacy application until the binary contents have
20 been loaded into the memory reference of the legacy application. The runtime
21 environment framework will create a new instance of the component identified by the
22 API and enable it to be populated with information. The component is populated using
23 object-oriented programming techniques by applications that use the present invention.

1 When the component is fully populated, the component will construct a binary data
2 layout suitable for the architecture and programming language of the legacy application
3 and present the binary data record to the legacy application using the interprocess
4 communications mechanism implemented within the present invention. When the
5 binary data is presented to the API framework of the present invention, the legacy
6 application will unblock with the contents of the data record populated with the values
7 programmed into the component.

8 **Figure 11** represents the high-level relationship between these three
9 frameworks. The workbench framework is an offline (or non-runtime) process that
10 maps the existing data structures from the existing source code into metadata that is
11 stored in a repository. The two online frameworks, the API and the component runtime
12 environment, interact programmatically using the metadata in the repository to bridge
13 any architectural differences between the system running the procedural code and the
14 system running the object-oriented code.

Brief Description of the Drawings

For a complete understanding of the present invention, reference should be made to the following detailed description in conjunction with the accompanying drawings.

Fig. 1 is a block diagram showing a computer system on which a legacy application and the runtime environment of the present invention reside as well as the flow of data between the frameworks.

Fig. 2 is a block diagram showing the flow of data from the source code representation to an architecture-neutral and language neutral representation.

Fig. 3 shows a sample legacy record definition in the form of a COBOL copybook.

Fig. 4 shows the language-neutral representation of a legacy record definition that corresponds to the COBOL copybook in Fig. 3.

Fig. 5 shows the source code in Java for the base class for all generated components.

Fig. 6 shows sample source code that implements the legacy record definition in Figures 3 and 4.

Fig. 7 shows the Unified Modeling Language description of the base class for all generated components shown in Fig. 5.

Fig. 8 shows the flow of information in and out of the workbench frameworks.

Fig. 9 shows the flow of a typical procedural program and one that specifically interacts with the invention.

1 **Fig. 10** shows the implementation of a programming interface to the present
2 invention using the C programming language.

3 **Fig. 11** shows the high-level relationship between the three frameworks of the
4 invention (the “Workbench”, the “API” and the “Component Runtime Environment”) and
5 the use of the metadata obtained from the workbench in the runtime environment.

6 **Fig. 12** shows a block diagram of the computer system on which the workbench
7 frameworks reside.

8 **Fig. 13** shows a partial listing of the `BinaryRenderingEngine` class that is used to
9 transform the underlying architecture-specific data fields to and from their
10 corresponding character values.

Detailed Description of the Preferred Embodiment

As described below, the present invention will be carried out on a single computer or by one or more computers in a computer network. Referring to **Figure 12**, the computer on which the workbench framework is hosted contains a processor **31**, memory **33** and an operating system **32**. Thus, for example, the computer used in the present invention is a personal computer or workstation that is running an Intel or RISC processor running an operating system such as Microsoft Windows NT, IBM OS/2, IBM AIX or Sun Microsystems Solaris. The workbench framework ("workbench") **37**, one of three frameworks, utilizes existing source record layout definitions **35** as represented by a human readable textual description of a record in the source code of a computer language. This is stored on a persistent medium such as a hard disk drive and manipulated via a text editor that is not part of the present invention. An existing COBOL copybook, an example of which is shown in **Figure 3**, or a PL/I record definition in the source code of an existing legacy application are examples of a source record definition. However, the source languages capable of being utilized by the present invention are not limited to COBOL or PL/I. The source record definition is processed by a lexical analyzer **Figure 2** capable of translating the language-specific representation of a record layout into a language-neutral and computer-architecture neutral representation of the data layout ("metadata"). This metadata is stored on a persistent storage medium **35 of Figure 12** and accessed and managed via the workbench. An example of metadata is shown in **Figure 4**. This corresponds to the COBOL copybook in **Figure 3**. The relationship between the COBOL copybook and the metadata is isomorphic. The <name> tag in the metadata translates directly into

1 the name of the variable in the original COBOL copybook. Preferably the lexical
2 analyzer implements an LALR(1) grammar, as is well-known in the art, but this is not a
3 requirement of the present invention. The preferred embodiment of the present
4 invention implements this translation to a language-neutral representation of the record
5 layout for reasons of portability and runtime performance although this translation step
6 is not a requirement of the present invention. As a less-preferred alternative to the
7 language-neutral representation of the metadata would be to simply use the record
8 layout from the original source code. However, it is anticipated that the performance
9 degradation as a result of this implementation would limit the practicality of directly
10 parsing the original-language source code at runtime. This would reduce the
11 requirements of the code in the workbench to managing a repository of source-code
12 definitions instead of managing a repository of language-neutral metadata.

13 The application programming interface ("API"), the second of three frameworks
14 of the present invention, is a language-specific and runtime environment-specific
15 software layer that enables the legacy applications programmer working with a specific
16 language within a specific runtime environment to access the facilities of the runtime
17 framework of the present invention. Referring to **Figure 1**, the API **22** represents the
18 programming interface that passes a reference (usually via a memory address or
19 "pointer") to the binary data within the legacy application to and from the interprocess
20 communications layer to utilize services of the runtime frameworks of the present
21 invention. An example of a programming interface has been built for C programmers
22 on an IBM S/390 mainframe system to access the runtime services of the present
23 invention. This example is presented in **Figure 10**. However, this is an example of

1 legacy runtime environments supported by the API framework. Other implementations
2 for other languages, operating systems and runtime environments are also supported
3 by the present invention.

4 Referring to **Figure 1**, the computer on which the legacy application is hosted
5 contains a processor **20**, memory **21** an operating system **30** and a language runtime
6 environment for the programming language in which the legacy application is written.
7 Thus, for example, the computer used in the present invention for hosting the legacy
8 application is a server class computer running an Intel, RISC or CISC processor
9 running an operating system such as Microsoft Windows NT, IBM OS/2, Compaq
10 OpenVMS, Sun Microsystems Solaris, IBM AIX, or IBM OS/390. The computer on
11 which the component runtime is hosted, contains also contains a processor **20**, memory
12 **21**, an operating system **30** and a language runtime environment for the programming
13 language in which the component runtime application is written. The language runtime
14 may be a Java Virtual Machine, or a more traditional runtime environment that ships
15 with the language compiler under which the component was developed, such as the
16 MSVCRT.DLL that ships with Microsoft Visual C++. The language runtime is used by
17 the present invention but should not be confused with the component runtime. Thus, for
18 example, the computer used in the present invention for hosting the component runtime
19 is a server class computer running an Intel, RISC or CISC processor running an
20 operating system such as Microsoft Windows NT, IBM OS/2, Compaq OpenVMS, Sun
21 Microsystems Solaris, IBM AIX, or IBM OS/390.

22 The legacy application and the component runtime may co-exist on the same
23 computer or they may exist on separate computers connected by a computer network.

1 Thus, referring to **Figure 1**, the interprocess communications **23** may be an
2 interprocess and inter-computer communications mechanism implemented as a
3 computer networking protocol. Thus, for example, the legacy application may be
4 communicating with the component runtime via an operating system-defined serial
5 interprocess communications mechanism such as named pipes or FIFOs, or they may
6 communicate via a networking protocol such as TCP/IP sockets, IBM SNA APPC
7 communications, or asynchronous serial communications.

8 The component runtime, the third of three frameworks of the present invention, is
9 the most complex of the three. The component runtime is responsible for two basic
10 operations. The first is the construction of components based on a reference to the
11 metadata constructed by the workbench. The second is the population of the contents
12 of that component in an architecture-neutral form and the emission of the contents of
13 the component in an architecture-specific binary form when communicating with the
14 legacy application. These two primary operations of the component runtime is best
15 understood by describing the sequence of events when binary data is to flow from the
16 legacy application to the runtime and similarly when binary data flows from the runtime
17 to the legacy application.

18 The present invention implements the following operational processes to
19 implement data flow from a legacy application to the component runtime. Referring to
20 **Figure 1**, the legacy application is running on a computer processor and has
21 constructed a representation of a business datum in a manner appropriate for the
22 application using facilities in the legacy application source code. The legacy
23 application calls the API **22** of the present invention with a reference (or "pointer") to

1 the architecture-specific binary data and an indication of the name of the record as
2 implemented by the API. The API **22**, sends the architecture-specific binary data and
3 the name of the legacy record to an interprocess communications mechanism **23**. This
4 interprocess communications is typically, but not limited to, a TCP/IP network
5 connection. The name of the legacy record is used as a reference to the metadata of
6 the legacy record created by the workbench.

7 The component runtime then passes the reference **24** to the metadata to the
8 component factory **25**. The component factory is part of the software that comprises
9 the component runtime. The responsibility of the component factory is to dynamically
10 construct the objects that manage the data being exchanged between the legacy
11 application and the object-oriented systems that interoperate with the objects
12 constructed by the component runtime. It implements the "Abstract Factory" design
13 pattern as described in "Design Patterns. Elements of Reusable Object-Oriented
14 Software" by Gamma, et. al. and is well-known to those skilled in the art. The
15 component factory determines if an object class description exists for the metadata. An
16 example of this using a component runtime in Java would be an object of type *Class*. If
17 the class for the object exists, the object class is used to construct a new instance of
18 the class that can represent the contents of the legacy application's binary record. If
19 there is no object class description available to the component factory, the metadata is
20 used by the component factory to generate human-readable, textual source code in the
21 object-oriented language implemented by the component runtime. Thus, for example, if
22 the Java language is used as the object-oriented language in which to implement the
23 component runtime, then the component factory would generate Java source code that

1 is appropriate to interpret the contents of the binary data within the legacy application's
2 record structure. The source code, in this example Java source code, is used as input
3 into a compiler implementing the language, in this example, a Java compiler. The
4 compiler will generate a binary class definition file. That class definition file is used to
5 construct a new instance of the class that is suitable for managing the binary data from
6 the legacy record.

7 The next step in the process is for the component runtime to pull the data from
8 the interprocess communications mechanism **23** and pass the data to the component
9 via the data feed **28** that is accessible by the generated component. The component
10 was generated by the component factory with the metadata for a specific legacy record
11 layout, so there is code generated to accommodate the architecture-specific binary
12 data stream. On a datum by datum basis, the architecture-specific binary data stream
13 is read by the specially-created component and converted to corresponding fields
14 within the generated component by a binary rendering engine. This includes the
15 transformation of textual information from one character encoding set, for example
16 EBCDIC to the character encoding of the component runtime, for example Unicode. It
17 also includes the transformation of little-endian architecture numbers to big-endian
18 architecture numbers using appropriate bit manipulation for that particular field.
19 Using the metadata example from **Figure 4**, the software component would have been
20 specifically constructed to perform the following series of operations, assuming that

1 metadata came from a machine on an IBM S/390 architecture and the component
2 runtime was written in the Java programming language:

- 3 1) Read 6 bytes (since the “size” attribute of the metadata indicates this is a
4 six-byte field) and convert from binary-coded-decimal (“BCD”) to an
5 integer value. Assign that number to the object field named “ID-
6 NUMBER”.
- 7 2) Read 6 bytes and convert from BCD to an integer value. Assign that
8 number to the object field named “PIN”.
- 9 3) Read 35 bytes and convert from EBCDIC to Unicode. Assign that
10 character string to the object field named “NAME”.
- 11 4) For each of the three elements in the array:
12 a) Read 25 bytes and convert from EBCDIC to Unicode.
13 Assign that string to the object field “ADDRESS”
14 within the array in the object.
- 15 5) Read 12 bytes and convert the values from EBCDIC to Unicode. Assign
16 that string to the object field named “PHONE-NUMBER”.
- 17 6) Read 11 bytes and convert the values from EBCDIC to Unicode. Assign
18 that string to the object field named “SOCIAL-SECURITY-NUMBER”.
- 19 7) Read 3 bytes and convert those three bytes to a float number. Assign
20 that float value to the object field named “GRADE-POINT-AVERAGE”.
- 21 8) Read 3 bytes and convert from a S/390 packed decimal format to an
22 integer. Assign that integer value to the object field named
23 “BALANCES/TUITION”.

- 1 9) Read 3 bytes and convert from a S/390 packed decimal format to an
2 integer. Assign that integer value to the object field named
3 "BALANCES/HOUSING".

4 The conversions from one data type to another is accomplished via the
5 BinaryRenderingEngine class that is part of the component runtime. A partial listing of
6 that class in the Java programming language is listed in **Figure 13**.

7 The process for sending architecture-specific data to the legacy application is
8 similar, but the process starts with the object-oriented code. The programmer
9 programming in the object-oriented language, for example Java, requests from the
10 component runtime that an object of a specific type is constructed. If the binary class
11 definition file exists for that type, a new instance is constructed with empty values. If
12 there is no object class description available to the component factory, the metadata is
13 used by the component factory to generate human-readable, textual source code in the
14 object-oriented language implemented by the component runtime. Thus, for example, if
15 the Java language is used as the object-oriented language in which to implement the
16 component runtime, then the component factory would generate Java source code that
17 is appropriate to construct an appropriate architecture-specific binary data stream to be
18 sent back to the legacy application. The object-oriented programmer populated the
19 fields within the component with values that are applicable to the business problem it
20 was designed to solve. When the object-programmer is ready to pass the information
21 to the legacy application, it will be passed to the component runtime. In the Java
22 component runtime example shown in **Figure 5**, this process will be invoked via the
23 write (OutputStream)method. Since the component was generated to match a specific

1 legacy record definition, its source code was generated to convert the values in the
2 component to the architecture-specific binary representation using the binary rendering
3 engine using the reverse process of steps described in the above 9-step example. This
4 binary data is passed through the data feed **28** to the interprocess communications
5 mechanism **23**. The API **22** unblocks from the legacy application's call to read the
6 information with the binary data underlying the reference that was given by the API **22**
7 to the legacy record definition. The legacy application is insulated from the
8 transformations to a different architecture and the manipulations from object-oriented
9 code that results from the operation of the component runtime framework of the present
10 invention.

11 Referring to the UML diagram in **Figure 7**, each component contains a binary
12 rendering engine that is responsible for converting the information within the
13 component to and from the architecture-specific binary data used by the legacy
14 application. This process is more complex than is initially apparent. The first
15 complexity comes from the data translation. There are multiple character
16 representations, such as single-byte ASCII and EBCDIC and the double-byte Unicode.
17 There are also differences in the byte ordering ("endian-ness") of binary integer
18 representations among processor architectures. For example, the Intel x86 processor
19 family represents the least significant byte of the integer first ("little-endian") whereas
20 most other processor architectures such as the Sun SPARC and the IBM S/390
21 processor architectures represent the most significant byte first ("big-endian"). Finally,
22 not all elementary data types are supported on all processor architectures. For
23 example, the IBM S/390 processor architecture implements binary coded decimal

(BCD) data as a basic data type. BCD is not supported natively on Intel or most RISC processors. The second form of complexity occurs in the way that language compilers generate the legacy record definitions. Most computer processors can perform arithmetic computer operations faster if the numbers on which they operate are aligned on an even machine word boundary, usually an even multiple of two, four or eight bytes. The compiler developers will exploit this performance improvement by inserting additional bytes ("filler bytes") in the record definition to ensure alignment on even boundaries. The generated component accommodates these additional filler bytes in addition to the translation of each datum. A partial listing of the BinaryRenderingEngine is included in **Figure 13**. It reveals part of the complexity that the present invention encapsulates so that applications programmers are not exposed to this level of architecture-specific binary representations.